NUMPY FUNDAMENTALS

What is NumPy?

- NumPy is the fundamental package for scientific computing in Python.
- It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and it provide fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.
- At the core of the NumPy package, is the ndarray object. This encapsulates n-dimensional arrays of homogeneous data types.

NumPy Arrays Vs Python Sequences:

- NumPy arrays have a fixed size at creation, unlike Python lists (which can grow dynamically).
- Changing the size of a ndarray will create a new array and delete the original.
- The elements in a NumPy array are all required to be of the same data type (homogeneous), and thus will be the same size in memory.
- NumPy arrays facilitate advanced mathematical and other types of operations on large numbers of data.
- Typically, such operations are executed more efficiently and with less code than is possible using Python's built-in sequences.
- A growing plethora of scientific and mathematical Python-based packages / libraries like TensorFlow, Pandas, Scikit-learn etc. are using NumPy arrays. though these typically support Python-sequence input, they convert such input to NumPy arrays prior to processing, and they often output NumPy arrays.

Creating NumPy Arrays:

- NumPy is used to work with arrays. The array object in NumPy is called ndarray.
- We can create a NumPy ndarray object by using the array() function.
- There are two ways to importing the NumPy module:
  - ~ import numpy as np
  - ~ from numpy import *

```
# Importing the NumPy library & rename as np
import numpy as np
a = np.array([1,2,3,4])
# It's output called as vector or 1D array
print('a :',a)
```

```
a : [1 2 3 4]
```

```
# Importing the NumPy library using 2nd way
from numpy import *
b = array([1,2,3,4])
print('b :',b)
```

```
b : [1 2 3 4]
```

Creating 2D NumPy Arrays:

- 2D array are represented as collection of rows and columns.
- In machine learning and data science NumPy 2D array known as a matrix.
- Specially use to store and perform an operation on input values.

```
# 2D array (it is matrix)
from numpy import *
b = array([[1,2,3],[4,5,6]])
print(b)
```

```
[[1 2 3]
 [4 5 6]]
```

Creating 3D NumPy Arrays:

- In machine learning and data science NumPy 3D array known as a tensor.
- Specially used to store and perform an operation on three-dimensional data like colour image.

Note:

There are so many ways to create numpy arrays depending on situations for that we use other function that are provided by numpy library.

```
# 3D array (also called as tensor)
c = array([[[1,2,3]],[[5,6,7]],[[8,9,10]]])
print(c)
```

```
[[[ 1  2  3]]

 [[ 5  6  7]]

 [[ 8  9 10]]]
```

Creating NumPy with different datatype using dtype:

- It refers to the data type of elements stored in a NumPy array.
- Allows you to create arrays with different data types, such as integers, floating-point numbers, and more.
- When creating NumPy arrays, you can indeed specify the data type of the elements using the dtype parameter.
- Syntax: np.array([1,2,3] , dtype=float)

```
# dtype
import numpy as np
f = np.array([1,2,3,4] , dtype=float)
c = np.array([1,2,3,4] , dtype=complex)
# non integer value consider as True
tf = np.array([1,2,0,4] , dtype=bool)
print('float :',f)
print('complex :',c)
print('boolean :',tf)
```

```
float : [1. 2. 3. 4.]
complex : [1.+0.j 2.+0.j 3.+0.j 4.+0.j]
boolean : [ True  True False  True]
```

np.ones() function:

- Returns a new array of given shape and dtype, where the element's value is set to 1 & Default dtype is float.
- It is useful in deep learning to initialize the weights values.
- Syntax: np.ones(shape, dtype=None, order='C')

```
# np.ones() function
import numpy as np
o = np.ones((2,4), dtype=int)
print(o)
```

```
[[1 1 1 1]
 [1 1 1 1]]
```

np.zeros() function:

- Returns a new array of given shape and type, where the element's value as 0.
- Default dtype is float .
- Syntax: np.zeros(shape, dtype=float, order='C')

```
# np.zeros() function
import numpy as np
z = np.zeros((3,3))
print(z)
```

```
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

np.arange() function:

- Used to create arrays containing regularly spaced values within a specified range.
- It generates values starting from start, up to (but not including) stop, with increments of step.
- If step is not provided, it defaults to 1.
- Syntax: np.arange(start, stop, step, dtype=None)

```
# np.arange()
import numpy as np
# with start and stop argument
e = np.arange(5,10)
print(e)
# with start, stop and step argument
e1 = np.arange(5,10,2)
print(e1)
```

```
[5 6 7 8 9]
[5 7 9]
```

np.reshape() function:

- Used to change the shape (dimensions) of an array without changing its data.
- Returns a new array with the same data but with a different shape.
- Useful when we want to convert a 1D array into a two-dimensional array or vice versa.
- It can also be used to create arrays with a specific shape, such as matrices and tensors.
- Syntax: np.reshape(arr, new_shape, order='C')
  - ~ a: input array.
  - ~ new_shape: shape of new array
  - ~ order: {'C', 'F', 'A'}, optional

```
# np.reshape() function
import numpy as np
r = np.arange(1,13)
re = np.reshape(r,(2,6))
print(re)
re1 = np.reshape(r,(6,2))
print(re1)
re3 = np.reshape(r,(3,4))
print(re3)
```

```
[[ 1  2  3  4  5  6]
 [ 7  8  9 10 11 12]]
[[ 1  2]
 [ 3  4]
 [ 5  6]
 [ 7  8]
 [ 9 10]
 [11 12]]
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

Note:
New array dimension product is equal to number of items that are present in inside the original array.

np.ones() function:

- Returns a new array of given shape and dtype, where the element's value is set to 1.
- Default dtype is float.
- It is useful in deep learning to initialize the weights values
- Syntax: np.ones(shape, dtype, order='C')

```
# np.ones() function
import numpy as np
o = np.ones((2,4), dtype=int)
print(o)
```

```
[[1 1 1 1]
 [1 1 1 1]]
```

np.zeros() function:

- Returns a new array of given shape and type, where the element's value as 0.
- Default dtype is float .
- Syntax: np.zeros(shape, dtype=float, order='C')

```
# np.zeros() function
import numpy as np
z = np.zeros((3,3))
print(z)
```

```
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

np.random.random() function:

- Used for generating random numbers.
- Here 1st random is class name and other one is method name follows OOP concept.
- Syntax: np.random.random(shape, dtype)

```
# np.random.random() function
import numpy as np
# creating default random varible between 0 to 1
r = np.random.random((3,3))
print(r)
```

```
[[0.32138088 0.05043471 0.29236916]
 [0.67078606 0.7525749  0.51963277]
 [0.85785698 0.07023304 0.84409173]]
```

np.linspace() function : (Linear/ linearly space)

- Returns evenly spaced numbers over a specified interval.
- Use for plotting the ML algorithm result.
- Syntax: np.linspace(start, stop, num=50, dtype=float, axis=0)
  - ~ start: starting value of the sequence
  - ~ stop: end value of the sequence
  - ~ num: number for spacing & default is 50
  - ~ axis: axis for evenly spaced numbers & default is 0.

```
# np.linspace() function
import numpy as np
ls = linspace(1,10,5)
print(ls)
```

```
[ 1.    3.25  5.5   7.75 10.  ]
```

np.identity() function:

- Returns a square identity matrix of size n x n means diagonally items are 1 and remain all numbers becomes 0's
- Syntax: np.identity(n, dtype=float)
  - ~ n: size of the identity matrix
  - ~ dtype: we can use another datatype

```
# np.identity() functioin
import numpy as np
i = np.identity(3, dtype=int)
print(i)
```

```
[[1 0 0]
 [0 1 0]
 [0 0 1]]
```

Attributes of NumPy Arrays:

- NumPy array is the most used construct of numpy in machine learning and deep learning.
- Let us look into some important attributes of this numpy array.

```
# Arrays for applying numpy arrays attributes
import numpy as np
a1 = np.arange(10)
a2 = np.arange(12, dtype=float).reshape(3,4)
a3 = np.arange(8).reshape(2,2,2)
print(a1)
print(a2)
print(a3)
```

```
[0 1 2 3 4 5 6 7 8 9]
[[ 0.  1.  2.  3.]
 [ 4.  5.  6.  7.]
 [ 8.  9. 10. 11.]]
[[[0 1]
  [2 3]]

 [[4 5]
  [6 7]]]
```

arr.ndim attribute:

- Returns the number of dimensions of a given numpy array.

```
# arr.ndim attribute
print(a1.ndim)
print(a2.ndim)
print(a3.ndim)
```

```
1
2
3
```

arr.shape attribute:

- Determine the dimensions of the array and returns a tuple of integers that represent the size of the array in each dimension.

```
# arr.shape attribute
print(a1.shape)
print(a2.shape)
# Made with two 2D array of shape 2, 2
print(a3.shape)
```

```
(10,)
(3, 4)
(2, 2, 2)
```

arr.size attribute:

- **Returns the total number of elements in the array.**

```
# arr.size attribute
print(a1.size)
print(a2.size)
print(a3.size)
```

```
10
12
8
```

arr.itemsize attribute:

- **Returns the size (in bytes) of each element in the array.**

```
# arr.itemsize attribute
print(a1.itemsize) # int32 = 4 bytes
print(a2.itemsize) # int64 = 8 bytes
print(a3.itemsize) # int32 = 4 bytes
```

```
4
8
4
```

arr.dtype attribute:

- **Returns the datatype of the elements in the array.**

```
# arr.dtype attribute
print(a1.dtype)
print(a2.dtype)
print(a3.dtype)
```

```
int32
float64
int32
```

Changing datatype using .astype() method:

- **Change the data type of the elements in the array.**
- **More useful in converting the float datatype reduction in integer value.**

```
# arr.astype() method
print(a2)
print('a2 dtype :',a2.dtype)
change_dtype = a2.astype(int32)
print(change_dtype)
print('a2 dtype :',change_dtype.dtype)
```

```
[[ 0.  1.  2.  3.]
 [ 4.  5.  6.  7.]
 [ 8.  9. 10. 11.]]
a2 dtype : float64
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
a2 dtype : int32
```

NumPy Array Operations:

Use for performing mathematical operations

```
# Creating the arrays for operations
from numpy import *
a1 = arange(12).reshape(3,4)
a2 = arange(12,24).reshape(3,4)
print(a1)
print(a2)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
[[12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
```

Scalar operations:

- **Scalar operation is an operation between a scalar value (a single number) and an array.**
- **It can be performed using arithmetic operators such as +, -, *, and /.**
- **Scalar value perform operation with each individual element in the array.**

```
# scalar operations, here 2 scalar value
from numpy import *
print(a1 + 2) # addition of each element
print(a1 * 2) # multiplication of each element
print(a1 ** 2) # exponent of each element
```

```
[[ 2  3  4  5]
 [ 6  7  8  9]
 [10 11 12 13]]
[[ 0  2  4  6]
 [ 8 10 12 14]
 [16 18 20 22]]
[[  0   1   4   9]
 [ 16  25  36  49]
 [ 64  81 100 121]]
```

Comparison / relational operations:

- **NumPy provides comparison operators such as ==, <, >, <=, >= etc. for comparing elements in two arrays.**
- **This operations return a boolean array with the same shape as the input arrays.**
  - **~ True indicates condition is satisfied**
  - **~ False indicates condition not satisfied**

```
# comparison operators
print(a1 >= 0)
print(a1 == 2)
```

```
[[ True  True  True  True]
 [ True  True  True  True]
 [ True  True  True  True]]
[[False False  True False]
 [False False False False]
 [False False False False]]
```

Vector operations:

- **A vector operation is an operation between two arrays of the same size.**
- **Vector operations can also be performed using arithmetic operators.**
- **When two arrays are added, the corresponding elements in each array are added together & also similarly for another operations.**

```
# vector Operations
print(a1 + a2) # addition
print(a1 * a2) # multiplication
```

```
[[12 14 16 18]
 [20 22 24 26]
 [28 30 32 34]]
[[  0  13  28  45]
 [ 64  85 108 133]
 [160 189 220 253]]
```

Note:

Vector operations can only be performed on arrays of the same shape. If the arrays have different shapes, NumPy will raise a ValueError.

NumPy Array Functions:

Some common NumPy array function that is use in machine learning and deep learning etc.

NumPy array mathematical functions:

Use for performing mathematical function

```
# Creating arrays for numpy mathematical function
import numpy as np
a1 = np.random.random((3,3))
a1 = np.round(a1*100)
print(a1)
```

```
[[58. 75. 51.]
 [81.  9. 58.]
 [13. 71. 68.]]
```

np.min() & np.max():

- **Return the minimum and maximum of element in array.**
- **But we can also use NumPy min and max to compute the minima and maxima of each column and rows.**
  - **~ column-wise represents axis=0**
  - **~ row-wise represents axis=1**

```
# min() & max()
print(np.min(a1))
print(np.max(a1))
```

```
9.0
81.0
```

```
# min() & max() with axis parameter
# column wise each minimum element
print('min :',np.min(a1, axis=0))
# row wise each maximum element
print('max :',np.max(a1, axis=1))
```

```
min : [13.  9. 51.]
max : [75. 81. 71.]
```

np.sum() function:

- Used to calculate the sum of elements in a NumPy array.
- Also used to find the sum of all elements in the array or along a specific axis of a multi-dimensional array.
  - ~ column-wise represents axis=0
  - ~ row-wise represents axis=1

```
# sum()
print('sum of array :', np.sum(a1))
# sum() with axis parameter
print('sum of cols :',np.sum(a1, axis=0))
print('sum of rows :',np.sum(a1, axis=1))
```

```
sum of array : 484.0
sum of cols : [152. 155. 177.]
sum of rows : [184. 148. 152.]
```

np.prod() function:

- Return the product of all element in an array.
- Along with the axis parameter to calculate the product along a specific axis of a multi-dimensional array.
- It is a common operation in various mathematical and statistical calculations.

```
# prod()
print('product of array :', np.prod(a1))
# prod() with axis parameter
print('product of cols :',np.prod(a1, axis=0))
print('product of rows :',np.prod(a1, axis=1))
```

```
product of array : 588742745338800.0
product of cols : [ 61074.  47925. 201144.]
product of rows : [221850.  42282.  62764.]
```

np.round() function:

- Used to rounds the elements of an array to the nearest integer or to a specified number of decimals.

```
# round()
import numpy as np
arr = np.array([1.23,2.49,4.51])
print(np.round(arr))
```

```
[1. 2. 5.]
```

np.ceil() function:

- Used to rounds the elements of an array up to the nearest integer.

```
# ceil()
import numpy as np
arr3 = np.array([1.23,2.49,4.11,1.00])
print(np.ceil(arr3))
```

```
[2. 3. 5. 1.]
```

np.floor() function:

- Used to rounds the elements of an array down to the nearest integer.

```
# floor()
import numpy as np
arr1 = np.array([1.23,2.49,4.51,4.80])
print(np.floor(arr1))
```

```
[1. 2. 4. 4.]
```

```
print(np.floor(np.random.random((2,3))*100))
```

```
[[73.  1.  0.]
 [ 1. 82. 34.]]
```

np.log() function:

- To calculate the natural logarithm of an array or a scalar.

```
import numpy as np
ar = np.arange(1,10).reshape(3,3)
# log()
print(np.log(ar))
```

```
[[0.         0.69314718 1.09861229]
 [1.38629436 1.60943791 1.79175947]
 [1.94591015 2.07944154 2.19722458]]
```

np.exp() function:

- To calculate the exponential of an array or a scalar.

```
import numpy as np
ar = np.arange(1,10).reshape(3,3)
# exp()
print(np.exp(ar))
```

```
[[2.71828183e+00 7.38905610e+00 2.00855369e+01]
 [5.45981500e+01 1.48413159e+02 4.03428793e+02]
 [1.09663316e+03 2.98095799e+03 8.10308393e+03]]
```

np.dot() function:

- Function takes two array arguments and returns their dot product.
- The dot product of two vectors and specifying the condition that they must have the same dimensionality.

```
# Creating two the arrays with same dimensions
import numpy as np
arr1 = np.arange(12).reshape(3,4)
arr2 = np.arange(12,24).reshape(4,3)
print(arr1)
print(arr2)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
[[12 13 14]
 [15 16 17]
 [18 19 20]
 [21 22 23]]
```

```
# dot()
print(np.dot(arr1,arr2))
```

```
[[114 120 126]
 [378 400 422]
 [642 680 718]]
```

NumPy array statistical functions:

Here, only a few functions related to statistics have been introduced. We will cover the remaining functions in the statistics session.

np.mean() function:

- Used to calculate the arithmetic mean or average of the elements in each array.
- The mean is the sum of all the values in the array divided by the total number of values.
- It is a common measure of central tendency.

```
# mean()
import numpy as np
arr = np.array([[2,5,11],[9,98,93],[17,21,40]])
print('mean :',np.mean(arr))
```

```
mean : 32.888888888888886
```

np.median():

- Used to calculate the median of the elements in an array.
- The median is the middle value of an array when it is ordered.
- It is a measure of central tendency that is less affected by outliers than the mean.

```
# median
print('median',np.median(arr))
```

```
median 17.0
```

np.var() function:

- Used to calculate the variance of the elements in an array.
- Variance is a measure of how much the values in a dataset vary from the mean.
- It gives you an idea of the spread or dispersion of the data points.

```
# var()
print('variance :',np.var(arr))
```

```
variance : 1230.9876543209875
```

np.std():

- Used to calculate the standard deviation of the elements in an array.
- The standard deviation is a measure of how much the values in a dataset deviate from the mean.
- It is another measure of the spread or dispersion of the data points, like variance.

```
# std()
data = np.array([5, 10, 15, 20, 25])
print("SD:", np.std(arr))
```

```
SD: 35.08543364875212
```

Indexing in NumPy:

- In NumPy, each element in an array is associated with a number. The number is known as an array index.
- NumPy array indexing refers to the process of accessing elements or subarrays within a NumPy array.
- In short, fetching the element from an array.

Note: Array start form 0 index.

1D Indexing in NumPy Array:

- NumPy array indexing is used to access values in the 1D & multi-dimensional arrays.
- Indexing is an operation, use this feature to get a selected set of values from a NumPy array.
- It just like normal indexing like list and, we can you positive or negative indexing.
  - ~ positive indexing: array start from 0 index position
  - ~ negative indexing: array start from end -1 index position
- Syntax: array[index_position]

```
import numpy as np
arr1 = np.arange(10)
print('array :',arr1)
# positive indexing
print(arr1[0],arr1[4],arr1[6],arr1[2])
# negative indexing
print(arr1[-1],arr1[-5],arr1[-3],arr1[-7])
```

```
array : [0 1 2 3 4 5 6 7 8 9]
0 4 6 2
9 5 7 3
```

2D Indexing in NumPy Array:

- 2D numpy arrays are like a table with rows and columns.
- For accessing elements, we need to specify the row index and column index of the element.
- Syntax: array[row_index , column_index_that_row]

```
import numpy as np
arr2 = np.arange(12).reshape(3,4)
print('2D numpy array:')
print(arr2)
print('Accessing elements:')
print(arr2[2,3], arr2[1,0], arr2[2,1])
```

```
2D numpy array:
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
Accessing elements:
11 4 9
```

Note: Array rows & cols start from 0 index.

3D Indexing in numpy array:

- 3D numpy arrays are like a table with rows and columns.
- For accessing elements, we need to specify the row index and column index of the element.
- Syntax: array[arr_index , row_index , column_index_of_row]

```
import numpy as np
arr3 = np.arange(8).reshape(2,2,2)
print(arr3)
```

```
[[[0 1]
  [2 3]]

 [[4 5]
  [6 7]]]
```

```
# 3D Indexing in numpy array
print(arr3[1,1,0],arr3[0,1,1],arr3[0,0,1])
```

```
6 3 1
```

Note: Array rows & cols start from 0 index.

Slicing in NumPy:

- NumPy array slicing is used to extract some portion of data from the actual array.
- NumPy slicing is slightly different.
- Slicing can be done with the help of (:).
- Syntax: array[start : stop : step]
  - ~ start: index by default considers as '0'
  - ~ stop: index considers as a length of the array.
  - ~ step: default is '1'.

1D Slicing in NumPy Aarry:

- For 1D numpy arrays we use basic slicing, step slicing and omitting the indices.

```
import numpy as np
arr = np.arange(10)
print(arr)
```

```
[0 1 2 3 4 5 6 7 8 9]
```

```
print('Slicing with start:',arr[6:])
print('Slicing with stop:',arr[:4])
print('Slicing with step:',arr[::3])
print('Slicing with start & stop:',arr[2:6])
print('Slicing with start,stop & step:',arr[1:8:2])
print('Negative start & stop slicing:',arr[-6:-1])
```

```
Slicing with start: [6 7 8 9]
Slicing with stop: [0 1 2 3]
Slicing with step: [0 3 6 9]
Slicing with start & stop: [2 3 4 5]
Slicing with start,stop & step: [1 3 5 7]
Negative start & stop slicing: [4 5 6 7 8]
```

2D Slicing in NumPy Array:

- A 2D NumPy array can be thought of as a matrix, where each element has two indices, row index and column index.
- To slice a 2D NumPy array, we can use the same syntax as for slicing a 1D NumPy array.
- The only difference is that we need to specify a slice for each dimension of the array and use comma ',' for separating the rows and columns.
- Syntax:
  array[row_start : row_stop : row_step , col_start : col_stop : col_step]
  - ~ row_start: specifies starting index
  - ~ row_stop: stopping index
  - ~ row_step: step size for the rows respectively
  - ~ col_start: specifies starting index
  - ~ col_stop: stopping index
  - ~ col_step: step size for the columns respectively

```
import numpy as np
arr = np.arange(12).reshape(3,4)
print(arr)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

```
# Slicing 2D array
print(arr[0,:]) # print row
print(arr[2,:]) # print 3rd index row
print(arr[:,3]) # print 4th index column
```

```
[0 1 2 3]
[ 8  9 10 11]
[ 3  7 11]
```

```
# slicing sub-arrays
print(arr[1:,1:3])
print(arr[::2,1::2])
print(arr[::2,::3])
print(arr[0:2,1:])
```

```
[[ 5  6]
 [ 9 10]]
[[ 1  3]
 [ 9 11]]
[[ 0  3]
 [ 8 11]]
[[1 2 3]
 [5 6 7]]
```

3D Slicing in NumPy Array:

- A 2D NumPy array can be thought of as a matrix, where each element has two indices, row index and column index.

```
# Slicing 3D numpy array
import numpy as np
arr = np.arange(27).reshape(3,3,3)
print(arr)
```

```
[[[ 0  1  2]
  [ 3  4  5]
  [ 6  7  8]]

 [[ 9 10 11]
  [12 13 14]
  [15 16 17]]

 [[18 19 20]
  [21 22 23]
  [24 25 26]]]
```

```
# print 2nd position element of 3D array
print(arr[1])
```

```
[[ 9 10 11]
 [12 13 14]
 [15 16 17]]
```

```
# print 1st elements of 0 index
print(arr[0, 1])
# print 2nd column of index 1
print(arr[1,:,1])
```

```
[3 4 5]
[10 13 16]
```

```
print(arr[2,1:,1:])
```

```
[[22 23]
 [25 26]]
```

```
print(arr[::2,0,::2])
```

```
[[ 0  2]
 [18 20]]
```

Reshaping in NumPy:

In reshaping we commonly use reshape() and transpose() but sometimes we need to use revel() function.

```
# creating array
import numpy as np
arr = np.arange(1,10).reshape(3,3)
print(arr)
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

np.ravel() function:

- Converting the n-dimensional array into flatten (1D) array.
- Syntax: arr.ravel() or np.ravel(arr)

```
print('Original 2D array :')
print(arr)
print('Conveting into 1D array using ravel() :')
print(arr.ravel())
```

```
Original 2D array :
[[1 2 3]
 [4 5 6]
 [7 8 9]]
Conveting into 1D array using ravel() :
[1 2 3 4 5 6 7 8 9]
```

np.transpose() or .T function :

- Applied on 2D arrays to swipe the rows and columns of an array.
- Using transpose() function or we can also use the short name .T to transpose a 2D array.
- Syntax: arr.transpose() or arr.T

```
# example of transpose()
print('Original 2D array :')
print(arr)
print('Transpose of 2D Array :')
print(arr.transpose()) # OR print(arr.T)
```

```
Original 2D array :
[[1 2 3]
 [4 5 6]
 [7 8 9]]
Transpose of 2D Array :
[[1 4 7]
 [2 5 8]
 [3 6 9]]
```

Iteration On NumPy Array:

Iterating means going through elements one by one.
As we deal with multi-dimensional arrays in numpy, we can do this using basic for loop of python.

Iteration On 1D NumPy Array:

- It will go through each element one by one.

```
import numpy as np
# For loop on 1D array :
arr = np.arange(10)
for ele in arr:
    print(ele, end=' ')
```

```
0 1 2 3 4 5 6 7 8 9
```

Iteration On 2D NumPy Array:

- It will go through all the rows.

```
import numpy as np
# For loop on 2D array
arr = np.arange(12).reshape(3,4)
for ele in arr:
    print(ele)
```

```
[0 1 2 3]
[4 5 6 7]
[ 8  9 10 11]
```

Iteration On 3D NumPy Array:

- It will go through all the 2-D arrays.

```
import numpy as np
# For loop on 3D array
import numpy as np
arr = np.array([[[1, 2],[4, 5]],[[7, 8],[10, 11]]])
for ele in arr:
    print(ele)
```

```
[[1 2]
 [4 5]]
[[ 7  8]
 [10 11]]
```

np.nditer() function:

- It is a NumPy function that provides an efficient way to iterate over elements of a NumPy array.
- It allows iterating over multiple arrays simultaneously and provides a number of optional arguments that can be used to customize the iteration process.

```
# nd.iter() function
import numpy as np
arr = np.array([[[1, 2],[4, 5]],[[7, 8],[10, 11]]])
for ele in np.nditer(arr):
    print(ele, end=' ')
```

```
1 2 4 5 7 8 10 11
```

Stacking in NumPy:

- Stacking is the concept of joining arrays in NumPy.
- Arrays having the same dimensions can be stacked.
- We can stack arrays along different axes using the functions.
  - ~ column-wise represents axis=0
  - ~ row-wise represents axis=1
  - ~ np.hstack() : horizontal stacking
  - ~ np.vstack() : vertical stacking
- Sometimes we have multiple data source means data come from databases, API and another data comes from web scrapping etc. so that data is similar data for multiple sources then we can stack the data for data analysis.

```
# creating two arrays for hstack() and vstack():
import numpy as np
arr1 = np.arange(12).reshape(3,4)
arr2 = np.arange(12,24).reshape(3,4)
```

Note: Shape/dimension of the array should be same

np.hstack() function:

- Horizontal stacking concatenates the arrays in sequence horizontally (column-wise).
- This function stacks arrays horizontally (along axis 1)
- Syntax: np.hstack((arr1, arr2))

```
# Ex. of np.hstack()
print(np.hstack((arr1,arr2)))
```

```
[[ 0  1  2  3 12 13 14 15]
 [ 4  5  6  7 16 17 18 19]
 [ 8  9 10 11 20 21 22 23]]
```

np.vstack() function:

- Vertical stacking means concatenates the arrays in sequence vertically (row-wise).
- This function stacks arrays vertically (along axis 0).
- Syntax: np.vstack((arr1, arr2))

```
# Ex. of np.vstack()
print(np.vstack((arr1, arr2)))
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
```

Splitting in NumPy:

- Splitting is reverse operation of stacking.
- We can split the arrays into sub-arrays of the same shape
  - ~ np.hsplit(): horizontal splitting
  - ~ np.vsplit(): vertical splitting.

```
# creating array for hsplit() and vsplit() :
import numpy as np
arr = np.arange(1,13).reshape(3,4)
print(arr)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

Note: Only used to split an array into sub-arrays of equal size

np.hsplit() function:

- hsplit() function is used to split a numpy array into multiple sub-arrays horizontally (column-wise).
- Pass the input array and the number of sub-arrays as arguments.
- Syntax: np.split(arr, sub_arrays_size)
  - ~ arr: input array
  - ~ sub_arrays_size: number for splitting the array

```
# Ex. of hsplit()
print(np.hsplit(arr, 2))
```

```
[array([[ 1,  2],
       [ 5,  6],
       [ 9, 10]]), array([[ 3,  4],
       [ 7,  8],
       [11, 12]])]
```

np.vsplit() function:

- vsplit() function is used to split a numpy array into multiple sub-arrays vertically (row-wise).
- Pass the input array and the number of sub-arrays as arguments.
- Syntax: np.vsplit(arr, sub_arrays_size)
  - ~ arr: input array
  - ~ sub_arrays_size: number for splitting the array

```
# Ex. of vsplit()
print(np.vsplit(arr, 3))
```

```
[array([[1, 2, 3, 4]]), array([[5, 6, 7, 8]]), array
([[ 9, 10, 11, 12]])]
```

ADAVANCED NUMPY

NumPy Arrays Vs Python List:

let's compare NumPy arrays and Python lists based on the factors you mentioned: speed, memory, and convenience.

Speed:

- **NumPy arrays are generally faster than Python lists for numerical operations due to their fixed data type and memory layout.**
- **NumPy operations are optimized for performance using low-level C libraries.**

```
# python list
a = [i for i in range(10000000)]
b = [i for i in range(10000000,20000000)]
import time
c = []
start = time.time()
for i in range(len(a)):
    c.append(a[i]+b[i])
print(time.time()-start)
```

```
4.092959642410278
```

```
# python numpy
import numpy as np
a = np.arange(10000000)
b = np.arange(10000000,20000000)
start = time.time()
c = a + b
print(time.time()-start)
```

```
0.10450100898742676
```

Memory:

- **NumPy arrays use less memory compared to Python lists, especially for large datasets, due to their efficient memory layout and data type specification.**

```
# python list
import sys
a = [i for i in range(10000000)]
print(sys.getsizeof(a))
```

```
89095160
```

```
# python numpy
import sys
# default float
a = np.arange(10000000,dtype=np.int32)
print(sys.getsizeof(a))
```

```
40000112
```

Convenience:

- **Writing code with NumPy is often more concise and intuitive for numerical operations compared to using plain Python lists.**

In summary, if we dealing with numerical computations and performance is crucial, NumPy arrays are a better choice due to their speed and memory efficiency. However, if you need a more flexible and versatile data structure, Python lists might be more convenient.

Fancy Indexing in NumPy:

- **It allowing us to use an array or a list of indices rather than using a slice or a single integer index.**
- **More advanced indexing and selection of elements from an array.**
- **To perform fancy indexing, we can use an array or a list of indices to select specific elements or subarrays from an array.**
- **More useful in pandas.**
- **Syntax:**
  - **~ Row-wise: array[[row_indices]]**
  - **~ Column-wise: array[: , [column_indices]]**

```
import numpy as np
arr = np.arange(20).reshape(5,4)
print(arr)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]
 [16 17 18 19]]
```

```
print(arr[[0,2,4]]) # row-wise
```

```
[[ 0  1  2  3]
 [ 8  9 10 11]
 [16 17 18 19]]
```

```
print(arr[:,[0,2,3]]) # column-wise
```

```
[[ 0  2  3]
 [ 4  6  7]
 [ 8 10 11]
 [12 14 15]
 [16 18 19]]
```

np.random.randint() function :

- **It is used to generate a random integer within a specified range and shape.**
- **Syntax: np.random.randint(low, high, size, dtype=int)**
  - **~ low: lowest integer to be drawn from the distribution & it is inclusive**
  - **~ high: If high is not None, one integer is drawn from the range [low, high). If high is None, one integer is drawn from the range [0, low).**
  - **~ size: shape of the output array**
  - **~ dtype: datatype of the output array**

```
import numpy as np
arr = np.random.randint(1,70,16).reshape(4,4)
print(arr)
```

```
[[ 4 48 43  9]
 [50 12 40 24]
 [38 65 67  7]
 [32  2 49 68]]
```

Note:

Output array dimension number product is equal to number of items that are present in inside the original array.

Boolean Indexing in NumPy:

- **It is way of selecting elements from an array based on a boolean condition.**
- **Boolean mask is a numpy array containing truth values (True/False) that correspond to each element in the array.**
- **Boolean masking allows for the filtering of values in numpy arrays.**

Note:

For condition we use any operator that is satisfied the boolean condition like relational or bitwise operator etc.

```
# find all numbers less than and equal 40
import numpy as np
# boolean mask condition
bool_mask = arr <= 40
print('Boolean mask :')
print(bool_mask)
print('Boolean condition :',arr[bool_mask])
```

```
Boolean mask :
[[ True False False  True]
 [False  True  True  True]
 [ True False False  True]
 [ True  True False False]]
Boolean condition : [ 4  9 12 40 24 38  7 32  2]
```

- **More examples of boolean condition:**

```
# 1.find out even numbers
print(arr[arr%2 == 0])

# 2.find all numbers greater than 50 and are even
# Here we use bitwise because of boolean
print(arr[(arr%2 == 0) & (arr>50)])

# 3.find all numbers not divisible by 7
print(arr[(arr%7 != 0)])
```

```
[ 4 48 50 12 40 24 38 32  2 68]
[68]
[ 4 48 43  9 50 12 40 24 38 65 67 32  2 68]
```

Broadcasting in NumPy:

- An array with a smaller shape is expanded to match the shape of a larger one, this is called broadcasting.
- The term broadcasting describes how numpy treats arrays with different shapes during arithmetic operations.
- Smaller array is "broadcast" across the larger array so that they have compatible shapes.
- Use in vectorization.

```
# same shape
import numpy as np
a = np.arange(6).reshape(2,3)
b = np.arange(6,12).reshape(2,3)
print(a)
print(b)
print('Addtion with same shape:')
print(a+b)
```

```
[[0 1 2]
 [3 4 5]]
[[ 6  7  8]
 [ 9 10 11]]
Addtion with same shape:
[[ 6  8 10]
 [12 14 16]]
```

```
# different shape
import numpy as np
a = np.arange(6).reshape(2,3)
b = np.arange(3).reshape(1,3)
print(a)
print(b)
print('Addtion with different shape:')
print(a+b)
```

```
[[0 1 2]
 [3 4 5]]
[[0 1 2]]
Addtion with different shape:
[[0 2 4]
 [3 5 7]]
```

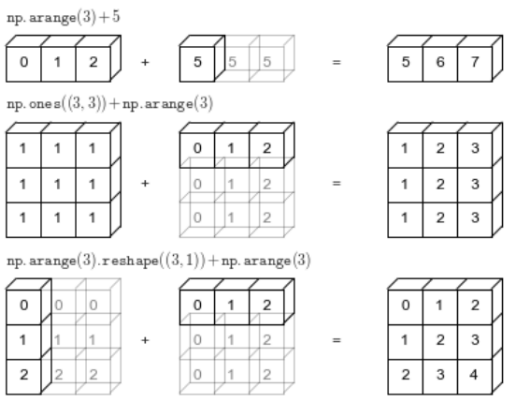
Rules for Broadcasting:

1.Make the two arrays have the same number of dimensions.

- If the numbers of dimensions of the two arrays are different, add new dimensions with size 1 to the head of the array with the smaller dimension.
  - ~ Ex.1: (3,2) & (3) → (3,2) & (1,3)
  - ~ Ex.2: (3,3,3) & (3) → (3,3,3) & (1,1,3)

2.Make each dimension of the two arrays the same size.

- If the sizes of each dimension of the two arrays do not match, dimensions with size 1 are stretched to the size of the other array.
- If there is a dimension whose size is not 1 in either of the two arrays, it cannot be broadcasted, and an error is raised.
  - ~ Ex.1: (3,2) & (3) → (3,2) & (1,3) → (3,2) & (3,3)
  - ~ Ex.2: (3,3,3) & (3) → (3,3,3) & (1,1,3) → (3,3,3) & (3,3,3)



- More examples to understanding broadcasting:

```
import numpy as np
n1 = np.arange(3) + 5
print(n1)
```

```
[5 6 7]
```

```
n1 = np.ones((3,3), dtype=int) + np.arange(3)
print(n1)
```

```
[[1 2 3]
 [1 2 3]
 [1 2 3]]
```

```
n1 = np.arange(3).reshape((3,1)) + np.arange(3)
print(n1)
```

```
[[0 1 2]
 [1 2 3]
 [2 3 4]]
```

```
n1 = np.arange(12).reshape(4,3)
n2 = np.arange(3)
print(n1+n2)
```

```
[[ 0  2  4]
 [ 3  5  7]
 [ 6  8 10]
 [ 9 11 13]]
```

- When shapes are not valid:

```
# ValueError: not be broadcast shapes (3,4) (3,)
import numpy as np
n1 = np.arange(12).reshape(3,4)
n2 = np.arange(3)
print(n1+n2)
```

```
# ValueError: not be broadcast shapes (4,3) (3,4)
import numpy as np
n1 = np.arange(12).reshape(3,4)
n2 = np.arange(12).reshape(4,3)
print(n1+n2)
```

```
# ValueError: not be broadcast shapes (4,4) (2,2)
import numpy as np
n1 = np.arange(16).reshape(4,4)
n2 = np.arange(4).reshape(2,2)
print(n1+n2)
```

Working With Mathematical Formulas in NumPy Array:

- For calculating the uncommon function that are not in build-in function in numpy library but we create our own function here let's discuss some mathematical formulas that are used in data science.

Sigmoid function:

- The sigmoid function is often used in logistic regression and artificial neural networks to introduce non-linearity.
- Calculating each item sigmoid
- Use in Deep learning and Machine learning algorithms
- Sigmoid range between 0 to 1
- Formula: 1 / (1+e^-x)

```
import numpy as np
def Sigmoid(array):
    return 1/(1+np.exp(-(array)))
a = np.arange(10)
s = Sigmoid(a)
print(s)
```

```
[0.5        0.73105858 0.88079708 0.95257413 0.98
201379 0.99330715
 0.99752738 0.99908895 0.99966465 0.99987661]
```

Mean squared error (MSE):

- In data science and machine learning to measure the average squared difference between the predicted values and the actual values.
- It is often used to assess the performance of regression models.
- It is loss function.

```
actual  = np.random.randint(1,50,25)
predicted  = np.random.randint(1,50,25)
```

```
print(predicted)
```

```
[29 22 42  9 13 38 39 40 44 46 46  7 42 29  4  8  7 2
6  8  6 12 10 13 39
 32]
```

```
print(actual)
```

```
[32 49 37 44 47 11 29  8 20 38  5 27 25 43 29  3 42
8 31  5 27 37 41 37
 38]
```

```
print(np.mean((actual - predicted)**2))
```

```
508.4
```

```
def MSE(atcual, predicted):
    return np.mean((actual-predicted)**2)
MSE(actual , predicted)
```

```
508.4
```

Working With Missing Values:

- **Dealing with missing values is a common task in data analysis and machine learning.**
- **Numpy provides a few ways to handle missing values.**

np.nan:

- **NumPy has a special value called NaN (Not a Number) that can represent missing values or undefined data in arrays.**

np.isnan() function :

- **Returns a boolean array where True indicates a NaN value.**

```
# working with missing values > np.nan
a = np.array([1,2,3,np.nan,4,np.nan,5])
print("a :",a)

# identfiying missing value using np.isnan
b = np.isnan(a)
print("boolean array :",b)
print('b :',a[~(np.isnan(a))]) # boolean indexing
```

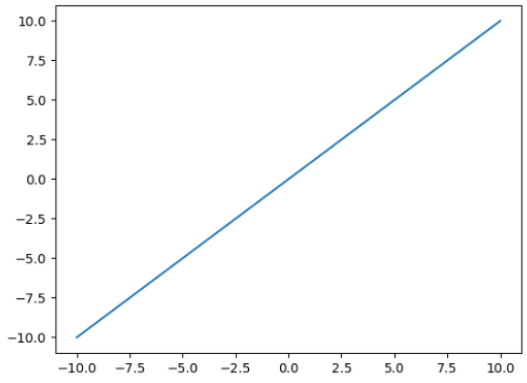
```
a : [ 1.  2.  3. nan  4. nan  5.]
boolean array : [False False False  True False  True
False]
b : [1. 2. 3. 4. 5.]
```

Plotting Graphs:

We can use NumPy in combination with Matplotlib to create and plot graphs.

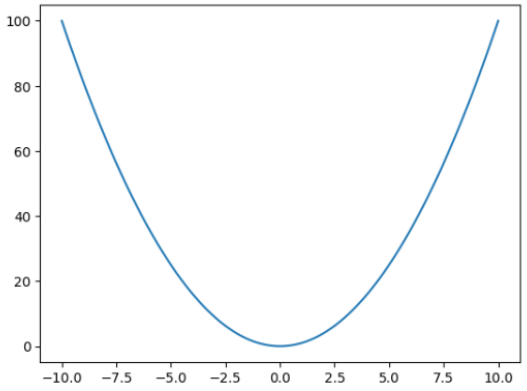
```
# plotting 2D plot
# x = y (straight line)
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(-10,10,100)
y = x
plt.plot(x,y)
```

```
[<matplotlib.lines.Line2D at 0x1f92229c160>]
```



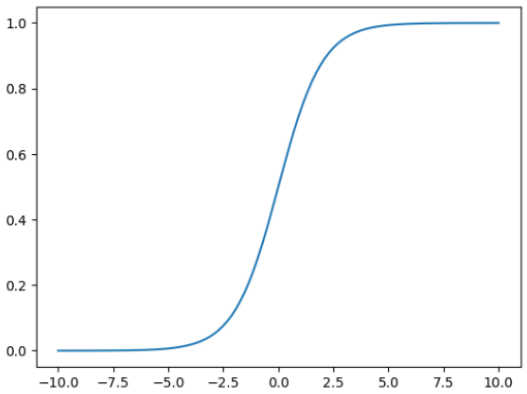

```
# parabola : The shape of this graph is a parabola.
# y = x^2
x = np.linspace(-10,10,100)
# applying scalar operation on x
y = x**2 # each element of x is square
plt.plot(x,y)
```

```
[<matplotlib.lines.Line2D at 0x1f922387550>]
```




```
# sigmoid graph
x = np.linspace(-10,10,100)
y = 1/(1 + np.exp(-x))
plt.plot(x,y)
```

```
[<matplotlib.lines.Line2D at 0x1f9224a9900>]
```



NUMPY TRICKS

np.sort() function:

- Return a sorted copy of an array.
  - ~ column-wise represents axis=0
  - ~ row-wise represents axis=1
- Syntax: np.argsort(arr, axis=-1, kind, order(optional))
  - ~ axis: default is -1 (the last axis)
  - ~ kind: Sorting algorithm. The default is 'quicksort'

```
import numpy as np
a = np.random.randint(1,20,size=15)
print(a)
b = np.random.randint(1,20,20).reshape(5,4)
print(b)
```

```
[ 5  7 14 15  6  7  9  5 11 19 13 15 16 15  7]
[[14 17  2 13]
 [10 14  5 11]
 [ 2 11 19  6]
 [14 19  7  7]
 [10  1 17  1]]
```

```
# ascending order(default)
asc = np.sort(a)
print("a:",asc)
# desending order
des = np.sort(a)[::-1]
print("d:",des)
```

```
a: [ 5  5  6  7  7  7  9 11 13 14 15 15 15 16 19]
d: [19 16 15 15 15 14 13 11  9  7  7  7  6  5  5]
```

```
c = np.sort(b, axis=0) # column-wise sort
r = np.sort(b, axis=-1) # row-wise sort
print('column-wise sort :')
print(c,'\n')
print('row-wise sort :')
print(r)
```

```
column-wise sort :
[[ 2  1  2  1]
 [10 11  5  6]
 [10 14  7  7]
 [14 17 17 11]
 [14 19 19 13]]

row-wise sort :
[[ 2 13 14 17]
 [ 5 10 11 14]
 [ 2  6 11 19]
 [ 7  7 14 19]
 [ 1  1 10 17]]
```

np.append() function:

- Appends values along the mentioned axis at the end of the array.
- Syntax: np.append(arr, values, axis)
  - ~ values: [array_like] values to be added in the arr

```
import numpy as np
arr = np.random.randint(1,20,size=15)
print(arr)
arr1 = np.random.randint(1,20,20).reshape(5,4)
print(arr1,'\n')
# Append elements in existing 1D array
append = np.append(arr,200)
print(append)
# Append column in 2D array
app = np.append(arr1,np.ones((arr1.shape[0],1)),axis=1)
print(app)
```

```
[ 9 11  2  9  6 19 10  6  4  9 11 12  1  8  9]
[[10  3 17 19]
 [ 8 12 11 13]
 [18  7  9  6]
 [ 8 15 12 19]
 [ 1  9  9  7]]

[  9  11   2   9   6  19  10   6   4   9  11  12   1   8
9 200]
[[10.  3. 17. 19.  1.]
 [ 8. 12. 11. 13.  1.]
 [18.  7.  9.  6.  1.]
 [ 8. 15. 12. 19.  1.]
 [ 1.  9.  9.  7.  1.]]
```

np.concatenate() function:

- Return a sequence of concatenate arrays along an existing axis.
- It is replacement of hstack() & vstack() function.
  - ~ column-wise represents axis=0
  - ~ row-wise represents axis=1
- Syntax: np.cancatenate([arr1, arr2..], axis)

```
import numpy as np
c = np.arange(6).reshape(2,3)
d = np.arange(6,12).reshape(2,3)
print(c)
print(d)
```

```
[[0 1 2]
 [3 4 5]]
[[ 6  7  8]
 [ 9 10 11]]
```

```
# default concatenate row wise
cc = np.concatenate((c,d),axis=0)
print(cc)
```

```
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
```

```
# concatenate column wise
cc = np.concatenate((c,d),axis=1)
print(cc)
```

```
[[ 0  1  2  6  7  8]
 [ 3  4  5  9 10 11]]
```

np.unique() function:

- Returns the sorted unique elements of an array.
- For example, consider a scenario where a single student registers for multiple courses. In this case, we aim to identify the unique users who have purchased the courses to ensure their usefulness.
- Syntax: np.unique(arr, axis)

```
import numpy as np
uni = np.array([1,2,3,3,4,4,5,5,6,6])
print("elements :",uni)
print('Unique Values :',np.unique(uni))
```

```
elements : [1 2 3 3 4 4 5 5 6 6]
Unique Values : [1 2 3 4 5 6]
```

np.expand_dims() function:

- Return an expanded the shape of an array.
- It is useful in 1D to 2D conversion or 3d to 4D conversion
- This function use in ML prediction and In DL to create batches of images.
- Syntax: np.expand_dims(arr, axis)

```
import numpy as np
arr = np.array([1,4,5,6,6])
print("1D array :",arr)
print('1D array shape', arr.shape)
# Expand the dimension of 1D array with column-wise
r = np.expand_dims(arr,axis=0)
print('2D :',r)
print('2D array shape:',r.shape)
```

```
1D array : [1 4 5 6 6]
1D array shape (5,)
2D : [[1 4 5 6 6]]
2D array shape: (1, 5)
```

```
# axis 1 is row-wise
print(np.expand_dims(arr,axis=1))
print('2D array shape:',a.shape)
```

```
[[1]
 [4]
 [5]
 [6]
 [6]]
2D array shape: (1, 5)
```

np.argmax() function:

- Returns the indices of the maximum values along an axis.
- Syntax: np.argmax(arr, axis)

np.argmin() function:

- Returns the indices of the minimum values along an axis.
- Syntax: np.argmin(arr, axis)

- Most of the time, both functions are used on 1D arrays in data science.
  - ~ column-wise represents axis=0
  - ~ row-wise represents axis=1

Note:

If same element occurrence in array while performing the functions, then prefer first occurrence element index.

```
import numpy as np
arr = np.array([1,2,3,4,4,20,5,5,0,6])
print("array :",arr)
```

```
array : [ 1  2  3  4  4 20  5  5  0  6]
```

```
# 1D array
print('max element index :',np.argmax(arr))
# 2D array
arr1 = np.arange(6,12).reshape(2,3)
print(arr1)
# column-wise (each column max index)
print('column max index:',np.argmax(arr1, axis=0))
# row-wise (each row max index)
print('row max index :',np.argmax(arr1, axis=1))
```

```
max element index : 5
[[ 6  7  8]
 [ 9 10 11]]
column max index: [1 1 1]
row max index : [2 2]
```

```
# 1D array
print('min element index :',np.argmin(arr))
# 2D array
arr1 = np.arange(6,12).reshape(2,3)
print(arr1)
# column-wise (each column min index)
print('column mix index:',np.argmin(arr1, axis=0))
# row-wise (each row min index)
print('row min index :',np.argmin(arr1, axis=1))
```

```
min element index : 8
[[ 6  7  8]
 [ 9 10 11]]
column mix index: [0 0 0]
row min index : [0 0]
```

np.where() function:

- Returns the indices of elements in an input array where the given condition is satisfied.
- Syntax: np.where(condition)

```
import numpy as np
arr = np.array([1,2,3,3,4,4,5,5,6,6])
print("array :",arr)

# Ex.Find all indices with value greater than 4
print(np.where(arr>4))

# Ex.Replace all values less 5 with 0
print(np.where(arr<5, 0, arr))
```

```
array : [1 2 3 3 4 4 5 5 6 6]
(array([6, 7, 8, 9], dtype=int64),)
[0 0 0 0 0 0 5 5 6 6]
```

np.percentile() function:

- Return compute the nth percentile of the given data (array elements) along the specified axis.
- Used in five summery to calculate interquartile.
- Syntax: np.percentile(arr, q, axis)
  - ~ q: percentages to compute, value must be between 0 to 100 inclusive.

```
p = np.arange(1,101)
# 0th percentile (minimum)
print(np.percentile(p, 0))
# 50th percentile (median)
print(np.percentile(p, 50))
# 100th percentile (maximum)
print(np.percentile(p, 100))
```

```
1.0
50.5
100.0
```

np.cumsum() function:

- Return the cumulative sum of the elements along a given axis.
- Syntax: np.cumsum(arr, axis)

np.cumprod() function:

- Return the cumulative product of elements along a given axis.
- Syntax: np.cumprod(arr, axis)
  - ~ column-wise represents axis=0
  - ~ row-wise represents axis=1

```
import numpy as np
arr = np.array([1,2,3,4,5])
print("array :",arr)
print('cumulative sum :',np.cumsum(arr))
arr1 = np.arange(1,7).reshape(2,3)
print(arr1)
print('cumulative sum :',np.cumsum(arr1))
# column-wise
print(np.cumsum(arr1,axis=0))
# row-wise
print(np.cumsum(arr1,axis=1))
```

```
array : [1 2 3 4 5]
cumulative sum : [ 1  3  6 10 15]
[[1 2 3]
 [4 5 6]]
cumulative sum : [ 1  3  6 10 15 21]
[[1 2 3]
 [5 7 9]]
[[ 1  3  6]
 [ 4  9 15]]
```

```
import numpy as np
arr = np.array([1,2,3,4,5])
print('cumulative prod :',np.cumprod(arr))
arr1 = np.array([[1, 2, 3], [4, 5, 6]])
print(np.cumprod(arr1))
# column-wise:cumulative product for each column
print(np.cumprod(arr1,axis=0))
# row-wise:cumulative product for each row
print(np.cumprod(arr1,axis=1))
```

```
cumulative prod : [  1   2   6  24 120]
[  1   2   6  24 120 720]
[[ 1  2  3]
 [ 4 10 18]]
[[  1   2   6]
 [  4  20 120]]
```

np.corrcoef() function:

- Return Pearson product-moment correlation coefficients.
- Pearson's r, which is a measure of the linear correlation between two variables.
- Correlation coefficient range generally in between 1 to -1.
- If coefficient is 0 means no change.
  - ~ Coefficient is 1 means both are positively correlated: means when experience increases then its salary also increases.
  - ~ Coefficient is -1 means both are negatively correlated: means if experience is less than its salary also less.
- Syntax: np.corrcoef(arr1, arr2)

```
import numpy as np
salary = np.array([20000,40000,25000,35000,60000])
experience = np.array([1,3,2,4,2])
print(np.corrcoef(salary,experience))
```

```
[[1.         0.25344572]
 [0.25344572 1.        ]]
```

np.histogram() function:

- Compute the histogram of a dataset.
- Useful in statistics.
- Syntax: np.histogram(arr, bins=10, range).
  - ~ bin: range of values of grouped together
  - ~ range: lower & upper range of bins

```
import numpy as np
data = np.array([1,2,2,3,0,3,3,4,0,4,5,6,7,6,7])
bins = [0,1,2,3,4,5,6,7]
print(np.histogram(data,bins))
```

```
(array([2, 1, 2, 3, 2, 1, 4], dtype=int64), array
([0, 1, 2, 3, 4, 5, 6, 7]))
```

```
# change in bins range
import numpy as np
arr = np.array([11,53,28,50,38,30,68,9,78,2,21])
print(arr)
bins=[0,40,80]
print(np.histogram(arr,bins))
```

```
[11 53 28 50 38 30 68  9 78  2 21]
(array([7, 4], dtype=int64), array([ 0, 40, 80]))
```

np.flip() function:

- reverses the order of array elements along the specified axis, preserving the shape of the array.
  - ~ column-wise represents axis=0
  - ~ row-wise represents axis=1
- Syntax: np.flip(arr, axis)

```
import numpy as np
# 1D array
arr = np.array([1,2,3,4,5])
print("Before flip :",arr)
print('After flip :',np.flip(arr))
```

```
Before flip : [1 2 3 4 5]
After flip : [5 4 3 2 1]
```

```
import numpy as np
arr1 = np.arange(0,4).reshape(2,2)
print(arr1)
```

```
[[0 1]
 [2 3]]
```

```
# 2D array
print(np.flip(arr1))
print(np.flip(arr1,axis=0)) # column-wise
print(np.flip(arr1,axis=1)) # row-wise
```

```
[[3 2]
 [1 0]]
[[2 3]
 [0 1]]
[[1 0]
 [3 2]]
```

np.isin() function:

- Used to determine whether each element in an input array is contained in a second array.
- It returns a Boolean array of the same shape as the input, where each element is True if the corresponding element in the input is found in the second array, and False otherwise.
- Use in panda's library.
- Syntax: np.isin(arr, test_arr)

```
import numpy as np
arr = np.array([2,6,5,2,100,3,4,5])
test_arr = [20,3,40,5,0,100]
print("array :",arr)
print(np.isin(arr,test_arr))
print('elements :',arr[np.isin(arr,test_arr)])
```

```
array : [  2   6   5   2 100   3   4   5]
[False False  True False  True  True False  True]
elements : [  5 100   3   5]
```

np.put() function:

- Replaces specific elements of an array with given values.
- Array indexed works on flattened array.
- Syntax: np.put(arr, [index_position], [values])

```
import numpy as np
ar = np.array([1,2,3,4,5])
print("Before array :",ar)
np.put(ar,[0,2,4],[90,50,33])
print('After putting values:',ar)
```

```
Before array : [1 2 3 4 5]
After putting values: [90  2 50  4 33]
```

Note: Permanent changes in array.

np.delete() function:

- Returns a new array with the deletion of sub-arrays along with the mentioned axis.
- Syntax: np.delete(arr, [index_position], axis)

```
import numpy as np
arr = np.array([1,2,3,4,5,6])
print("array :",arr)
print('Del specific item:',np.delete(arr,0))
print('Del multiple items:',np.delete(arr,[4,5]))
```

```
array : [1 2 3 4 5 6]
Del specific item: [2 3 4 5 6]
Del multiple items: [1 2 3 4]
```

np.clip() function:

- Used to Clip (limit) the values in an array.
- Useful in certain scenarios of machine learning and deep learning.
- Syntax: np.clip(arr, a_min, a_min)

```
import numpy as np
arr = np.array([6,27,6,6,84,35,57,59,57,38,51,46])
print('array:',arr)
print(np.clip(arr, a_min=20, a_max=60))
```

```
array: [ 6 27  6  6 84 35 57 59 57 38 51 46]
[20 27 20 20 60 35 57 59 57 38 51 46]
```

Set functions in NumPy:

Here we discuss some useful set function to perform set operations.

```
# Creating the two arrays
import numpy as np
arr1 = np.array([1,2,3,4,5])
arr2 = np.array([3,4,5,6,7])
print('arr1 :',arr1)
print('arr2 :',arr2)
```

```
arr1 : [1 2 3 4 5]
arr2 : [3 4 5 6 7]
```

np.union1d() function:

- Return the unique, sorted array of values that are in either of the two input arrays.
- Syntax: np.union1d(arr1, arr2)

```
# np.union1d()
print(np.union1d(arr1,arr2))
```

```
[1 2 3 4 5 6 7]
```

np.intersect1d() function:

- Return the sorted, unique values that are in both of the input arrays.
- Syntax: np.intersect(arr1, arr2)

```
# np.intersect1d()
print(np.intersect1d(arr1,arr2))
```

```
[3 4 5]
```

np.setdiff1d() function:

- Return the unique values in ar1 that are not in ar2.
- Syntax: np.setdiff1d(arr1, arr2)

```
# np.setdiff1d()
print(np.setdiff1d(arr1,arr2)) # for arr1
print(np.setdiff1d(arr1,arr2)) # for arr2
```

```
[1 2]
[1 2]
```

np.setxor1d() function:

- Return the sorted, unique values that are in only one (not both) of the input arrays.
- Syntax:
  np.setxor1d(arr1, arr2)

```
# np.setxor1d()
print(np.setxor1d(arr1,arr2))
```

```
[1 2 6 7]
```

np.in1d() function:

- Returns a boolean array the same length as ar1 that is True where an element of ar1 is in ar2 and False otherwise.
- Syntax: np.in1d(arr1, arr2)

```
# np.in1d()
print(np.in1d(arr1,3))
print(np.in1d(arr1,[1,3,4]))
```

```
[False False  True False False]
[ True False  True  True False]
```

EXTRA NUMPY FUNCTION THAT ARE USE IN TASK

np.tile() function:

- Repeating an array by repeating an input array multiple times along specified dimension.
- Useful when you want to create larger arrays by tiling or repeating smaller arrays.
- Syntax: np.tile(arr, reps)
  - ~ arr: input array that you want to repeat
  - ~ reps: a tuple specifying the number of times you want to repeat

```
import numpy as np
arr1 = np.array([1, 2])
arr2 = np.array([[1, 2], [3, 4]])
print('cols repeats 3 times:',np.tile(arr1, 3))
print('Repeats rows and cols by using tuple:')
# 2 is row & 3 is cols
print(np.tile(arr2, (2, 3)))
```

```
cols repeats 3 times: [1 2 1 2 1 2]
Repeats rows and cols by using tuple:
[[1 2 1 2 1 2]
 [3 4 3 4 3 4]
 [1 2 1 2 1 2]
 [3 4 3 4 3 4]]
```

np.unravel_index() function:

- Converts a flat index or array of flat indices into a tuple of coordinate arrays.
- Syntax: np.unravel_index(indices, shape)

np.flatten() function:

- Return 1D array
- Syntax: np.flatten(arr)

np.nan_to_num() function:

- Replace NaN values with a specified value, which in this case would be the mode of the non-NaN values.
- Missing value concept
- Syntax: np.nan_to_num(arr, arr2)
  - ~ arr: input array
  - ~ nan: a default is 0 or we can replace nan value

```
import numpy as np
arr = np.array([5, np.nan, 1, np.nan])
print('Before :',arr)
# default nan value is 0
print('After :',np.nan_to_num(arr, nan=0))
```

```
Before : [ 5. nan  1. nan]
After : [5. 0. 1. 0.]
```

np.broadcast_to() function:

- It allows you to create a new array with a specified shape by broadcasting the original data to that shape.
- Useful when you want to perform element-wise operations on arrays with different shapes, but compatible dimensions.
- Syntax: np.broadcast_to(arr, shape)
  - ~ arr: array to broadcast
  - ~ shape: shape of the desired array.

```
import numpy as np
arr = np.array([1,2,3])
print('Original array :', arr)
print('Broadcated array:')
print(np.broadcast_to(arr, (3,3)))
```

```
Original array : [1 2 3]
Broadcated array:
[[1 2 3]
 [1 2 3]
 [1 2 3]]
```

np.broadcast() function:

- Produce an object that mimics broadcasting.
- "Mimics broadcasting" means that NumPy makes it appear as if the arrays have been expanded to a common shape, allowing you to perform element-wise operations without manually duplicating data.
- Syntax: np.broadcast(arr1, arr2, ...)

```
import numpy as np
a = np.array([[1, 2], [3, 4]])
b = np.array([10, 20])
result = a + b  # Broadcasting happens here
print(result)
```

```
[[11 22]
 [13 24]]
```

np.empty() function:

- Return a new array of given shape and type, with random values
- Syntax: np.empty(shape, dtype=float)

```
import numpy as np
b = np.empty(2, dtype=int)
print("Matrix b : \n", b)
a = np.empty([2, 2])
print("\nMatrix a : \n", a)
c = np.empty([3, 3])
print("\nMatrix c : \n", c)
```

```
Matrix b :
 [10 20]

Matrix a :
 [[0.00000000e+000 1.15694187e-311]
 [1.15694187e-311 1.15694187e-311]]

Matrix c :
 [[0.00000000e+000 0.00000000e+000 0.00000000e+000]
 [0.00000000e+000 0.00000000e+000 7.74694933e-321]
 [1.15648771e-311 2.00485144e-307 0.00000000e+000]]
```

Note:

Empty, unlike zeros, does not set the array values to zero, and may therefore be marginally faster.

np.any() function:

- Tests whether any elements in a given array or along a specified axis evaluate to True.
- It returns a single Boolean value or an array of Boolean values, depending on the input.
- Syntax: np.any(arr, axis, keepdims)

```
import numpy as np
# if any element in the entire array is True
arr1 = np.array([False, False, False])
print('arr1:',np.any(arr1))
# if any element along a specific axis is True
arr2 = np.array([[False, True, False],
                 [False, False, False]])
print('arr2:',np.any(arr2, axis=0))
# Using keepdims
arr3 = np.array([[False, True, False],
                 [False, False, False]])
print('arr3:',np.any(arr3, axis=1, keepdims=True))
```

```
arr1: False
arr2: [False  True False]
arr3: [[ True]
 [False]]
```

np.argsort() function:

- Returns the indices that would sort an array and also sorted along with axis.
- Syntax: np.argsort(arr, axis=-1, kind , order)
  - ~ axis: default is -1 (the last axis)
  - ~ kind: sorting algorithm. The default is 'quicksort'

```
import numpy as np
arr = np.array([3, 1, 4, 1, 5, 9, 2, 6])
print(np.argsort(arr))
```

```
[1 3 6 0 2 4 7 5]
```

```
import numpy as np
arr = np.array([[3, 1, 4],[1, 5, 9],[2, 6, 5]])
print(arr)
print("Sorted indices along axis 0:")
print(np.argsort(arr, axis=0))
print("Sorted indices along axis 1:")
print(np.argsort(arr, axis=1))
```

```
[[3 1 4]
 [1 5 9]
 [2 6 5]]
Sorted indices along axis 0:
[[1 0 0]
 [2 1 2]
 [0 2 1]]
Sorted indices along axis 1:
[[1 0 2]
 [0 1 2]
 [0 2 1]]
```

np.around() function:

Syntax: np.around(arr1, arr2)

EXTRA NUMPY FUNCTIONS:

np.random.seed() function:

- **Save the state of a random function.**
- **The value in the NumPy random seed saves the state of randomness.**
- **For i.e., if we call the seed function using value 1 multiple times, the computer displays the same random numbers.**
- **Syntax:** np.random.seed(seed_value)

```
import numpy as np
# wihtout seed_value
np.random.seed()
np.random.randint(1,100,12).reshape(3,4)
```

```
array([[22, 55, 79, 63],
       [33, 81, 45, 69],
       [10, 38, 74, 46]])
```

```
# save the random state at position 0
np.random.seed(0)
np.random.randint(1,100,12).reshape(3,4)
```

```
array([[45, 48, 65, 68],
       [68, 10, 84, 22],
       [37, 88, 71, 89]])
```

```
# save the random state at position 1
np.random.seed(1)
np.random.randint(1,100,12).reshape(3,4)
```

```
array([[38, 13, 73, 10],
       [76,  6, 80, 65],
       [17,  2, 77, 72]])
```

np.random.shuffle() function:

- **We can get the random positioning of different integer values in the numpy array or we can say that all the values in an array will be shuffled randomly.**
- **Syntax:** np.random.shuffle(x)
  - **x: It is a sequence you want to shuffle such as list.**

```
import numpy as np
a = np.array([12,41,33,67,89,100])
print(a)
```

```
[ 12  41  33  67  89 100]
```

```
# after applying the np.random.shuffle() function
np.random.shuffle(a)
```

```
a
```

```
array([ 67, 100,  41,  12,  33,  89])
```

Note: **Permanent changes into the array or any sequence.**

np.random.choice() function:

- **We can get the random samples of one dimensional array and return the random samples of numpy array.**
- **Syntax:** np.random.choice(a, size, replace=True)
  - **a : 1D array of numpy having random samples.**
  - **size : output shape of random samples of numpy array.**
  - **replace: whether the sample is with or without replacement.**

```
import numpy as np
a = np.array([12,41,33,67,89,100])
np.random.choice(a,5)
```

```
array([12, 41, 67, 89, 33])
```

```
np.random.choice(a,4)
```

```
array([100, 100,  12,  33])
```

```
# to stop repeated number in choice
np.random.choice(a,4,replace=False)
```

```
array([ 33, 100,  67,  41])
```

np.swapaxes() function:

- Interchange two axes of an array.
- Syntax: np.swapaxes(arr, axis1, axis2)
  - ~ arr : input array.
  - ~ axis1 : first axis.
  - ~ axis2 : second axis.

```
#Example
x = np.array([[1,2,3],[4,5,6]])
print(x)
print(x.shape)
print("Swapped")
x_swapped = np.swapaxes(x,0,1)
print(x_swapped)
print(x_swapped.shape)
```

```
[[1 2 3]
 [4 5 6]]
(2, 3)
Swapped
[[1 4]
 [2 5]
 [3 6]]
(3, 2)
```

Note: It is not same as reshaping

np.random.uniform()

- Draw samples from a uniform distribution in rangge [low - high); high not included.
- Syntax: np.random.uniform(low, high, size=None)
  - ~ low: lower bound of sample; default value is 0
  - ~ high: upper bound of sample; defalut value is 1.0
  - ~ size: shape of the desired sample. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.
- Return the random samples as numpy array.
- Whenever we need to test our model on uniform data and we might not get truly uniform data in real scenario, we can use this function to randomly generate data for us.

np.repeat()

- Repeat elements of an array. `repeats` parameter says no of time to repeat.
- Syntax: np.repeat(a, repeats, axis)
  - ~ a: Input array.
  - ~ repeats: the number of repetitions for each element. repeats is broadcasted to fit the shape of the given axis.
  - ~ axis : 0 or 1

np.count_nonzero():

- This function counts the number of non-zero values in the array. https://numpy.org/doc/stable/reference/generated/numpy.count_nonzero.html
- Returns number of non-zero values in the array along a given axis. Otherwise, the total number of non-zero values in the array is returned.
- Syntax: np.count_nonzero(arr, axis, keepsdims)
  - ~ arr : the array for which to count non-zeros.
  - ~ axis : axis or tuple of axes along which to count non-zeros. Default is None, meaning that non-zeros will be counted along a flattened version of arr.
  - ~ keepdims : If this is set to True, the axes that are counted are left in the result as dimensions with size one.

np.allclose() function:

- Returns True if two arrays are element-wise equal within a tolerance. The tolerance values are positive, typically very small numbers.
- The relative difference `(rtol * abs(b))` and the absolute difference `atol` are added together to compare against the `absolute difference` between `a` and `b`.
- If the following equation is element-wise True, then `allclose` returns `True`.
- absolute(a - b) <= (atol + rtol * absolute(b))
- Syntax : numpy.allclose(arr1, arr2, rtol, atol, equal_nan=False)
  - ~ arr1 : input 1st array.
  - ~ arr2 : input 2nd array.
  - ~ rtol : the relative tolerance parameter.
  - ~ atol : the absolute tolerance parameter.
  - ~ equal_nan : whether to compare NaN's as equal.If True, NaN's in arr1 will be considered equal to NaN's in arr2 in the output array.
- Returns True if the two arrays are equal within the given tolerance, otherwise it returns False.